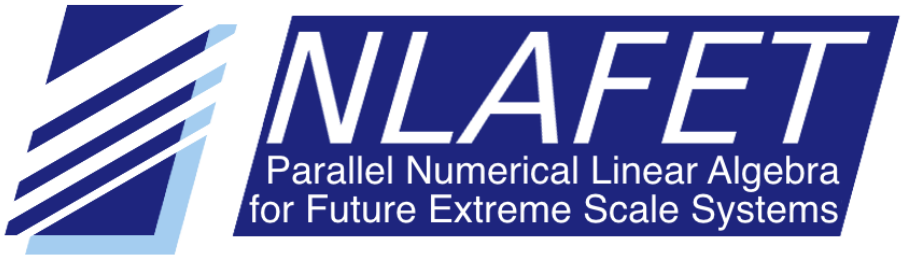

H2020–FETHPC–2014: GA 671633

D6.4

An Off-line Autotuning Framework based on Heuristic Search

March 2017

Document information

Scheduled delivery	2017-04-31
Actual delivery	(not yet delivered)
Version	0.1
Responsible partner	UNIMAN

Dissemination level

CO — Confidential

Revision history

Date	Editor	Status	Ver.	Changes
2017-03-27	Samuel Relton	Draft	0.1	Initial version of document produced.

Author(s)

Negin Baghepour (UNIMAN)
Samuel Relton (UNIMAN)
Mawussi Zounon (UNIMAN)

Internal reviewers

First Last (UMU/UNIMAN/STFC/INRIA)
First Last (UMU/UNIMAN/STFC/INRIA)

Contributors

Copyright

Acknowledgements

This project has received funding from the *European Union's Horizon 2020 research and innovation programme* under the grant agreement number 671633.

Table of Contents

1 Introduction **3**

2 Review of existing software **4**

3 Applying OpenTuner within NLAFET **5**

4 Curve fitting **6**

5 Performance results **8**

6 Conclusions **10**

List of Figures

1 The optimal block size for DGEMM is independent of the matrix size so a constant model is most appropriate. On this architecture the constant is 304. 6
2 The optimal block size for DGETRF appears to depend linearly on the matrix size. In this model the intercept is 92.25 with a gradient of 0.008. . 7
3 The optimal block size for SPOTRF appears to depend logarithmically on the matrix size. The least-squares fit provides the block size function $y = -898 + 135\log(x)$. 7
4 The optimal block size for DPOTRF appears to be a step function. The three constants in this step function are 124, 291, and 487. 8
5 Tuned and untuned performance for the GEMM kernel. Single precision is on the left and double precision on the right. 8
6 Tuned and untuned performance for Cholesky factorization. Single precision is on the left and double precision on the right. 9
7 Tuned and untuned performance for LU factorization. Single precision is on the left and double precision on the right. 9
8 Tuned and untuned performance for QR factorization. Single precision is on the left and double precision on the right. 10

1 Introduction

In most pieces of scientific software there are a number of parameters that can be tweaked to obtain better performance. These can take a variety of forms including continuous variables (e.g. a parameter in a model of some data), integer variables (e.g. the number of threads to spawn in a parallel section of code), or even categorical variables (e.g. on/off switches for compiler flags).

In this document we discuss, with a specific focus on the needs of the NLAFET project, how to optimise such parameters via autotuning. In particular we concentrate on off-line autotuning; meaning that the optimisation is performed before running the user runs the software on their own problems. An alternative approach is on-line autotuning, where the parameters are allowed to vary during the execution of a large problem. This latter approach will be covered in other NLAFET deliverables.

To begin, let us explain the specific needs of the NLAFET project with regards to off-line autotuning. The NLAFET project aims to deliver software for a large number of linear algebra problems, on both shared and distributed memory architectures, making use of runtime systems such as OpenMP, StarPU, and ParSec, for example.

Many of the algorithms employed to solve these linear algebra problems involve decomposing a large matrix into smaller "blocks". These blocks are then dealt with by separate cores (or nodes in the distributed memory setting) and are later combined to give the final solution to the problem. This approach is the key idea behind the PLASMA and MAGMA projects [1] and has been incorporated into many other libraries such as Intel MKL. In terms of autotuning, this means that positive integers controlling the size of the blocks must be chosen to maximize performance. Note that the optimal block size may be different for each linear algebra routine. For example, performing a matrix multiplication may required a different block size than an LU factorization.

Other algorithms, especially for sparse matrices, tend to be iterative in nature. In particular some algorithms have two layers of iteration (an inner and outer iteration) and the tolerance of each iteration can differ. Increasing the error tolerance of the iterations leads to a faster solution, sometimes at the expense of accuracy, making these values continuous tunable parameters.

Finally, all these algorithms need to compiled across a number of different architectures with different compilers, various numbers of cores, various number of GPUs etc. The compiler flags used during the compilation can have a dramatic effect on the overall performance. For example, whether or not vectorization is enabled when compiling with GCC will have a large impact on performance. Similarly using the flags -O1, -O2, or -O3 during compilation can have a dramatic impact on overall runtime. Further examples include parameters to select the runtime scheduling strategy in StarPU etc. These categorical parameters also need to be optimized to achieve the best possible performance.

The large number of parameters to optimise leads to a combinatorial explosion of possible options. Furthermore, due to the variety of parameter types that need to be optimized (integer, continuous, and categorical) a very general optimization routine is required. We believe that genetic algorithms are well-suited to optimise such high-dimensional problems with various parameter types. Currently, many pieces of software attempt to perform a "grid-sweep" which tries all (or a representative sample) of the parameter combinations before selecting the best. The grid-sweep will act a baseline for us to test the efficiency of other optimisation strategies considered.

To test our choice of autotuning strategy, we will show the results when this strat-

egy is employed to tune the PLASMA library for dense linear algebra over a range of architectures.

The rest of this report is organised as follows. In section 2 we compare a number of existing autotuning approaches, aiming to determine which one is most suitable for the NLAFET project. In section 3 we give more details on how the chosen strategy can be applied to linear algebra software by combining it with a Lua interface. Then in section 4 we analyze some typical behaviours exhibited by the block size and show how fitting curves to the block size data allows us to select near-optimal block sizes for all sizes of input matrix. In section 5 we show how our tuning strategy improves the performance of PLASMA for a number of PLASMA routines before giving some concluding remarks in section 6.

2 Review of existing software

There are a number of off-line autotuning frameworks already available from the HPC community which are largely split into two categories:

- those which require modification to the source code, and,
- those which interface with the source code via, for example, a configuration file.

The primary advantages of frameworks that modify the source code directly (such as the Periscope Tuning Framework [3], [4]) are that the developer has fine-grain control over specifically what is being optimised and that different sections of the code can be optimised independently; possibly speeding up the autotuning process. On the other hand, the overall code must be linked up to the tuning library which increases the number of dependencies of the software. Issues can also arise in the future if the chosen autotuning framework becomes unsupported due to lack of funding, or fails to compile on new architectures.

Performing autotuning through some intermediary interface has one distinct advantage over the previous approach: it is more modular and flexible. This separation between the computational routines and the autotuning means that, if our chosen optimiser fails to run on a certain architecture it can easily be replaced by another. To downside to this approach is that different parts of the software cannot be tuned independently: the entire algorithm must be re-run to try a new set of parameters.

Since the NLAFET project targets multiple architectures, including upcoming ARM-based HPC machines that are not readily available at present, we prefer to take the second approach and perform autotuning through configuration files. In order to maximize the number of architectures where our autotuning can be performed, we would like our optimiser to be written in a language such as C or Python, which are supported in all major operating systems.

Although there are a large number of autotuning frameworks available for download, the only one we have found that fits all of the criteria is OpenTuner [2]. OpenTuner is a Python module with minimal dependencies that can be used to optimise all the quantities mentioned previously. Particularly interesting is its use of ensemble optimisation: a large number of algorithms (including genetic optimisation and simulated annealing etc.) are fed into a multi-armed bandit model which detects the algorithms that are performing well on the current problem and gives them a larger share of the optimisation time. More detail on the specifics can be found in the reference above.

3 Applying OpenTuner within NLAFET

Now that we have decided on a tuning approach, this section describes in more detail how we plan to perform tuning withing NLAFET, using the PLASMA project (for dense linear algebra) as an example.

Overall, we aim to use optimization software such as OpenTuner (or even a full grid-sweep) to determine optimal values of the block size for a range of different matrix shapes and sizes. Once these values have been found, we need to inform PLASMA which block size is appropriate for each matrix via an intermediary interface. However, instead of using a simple configuration file we opt to use a Lua script.

Lua is a lightweight, embeddable, and open-source scripting language that is ideal for making small extensions to larger software projects. In this scenario, we can use Lua to return an "optimal" block size for matrices with sizes that we haven't tested by interpolating the values seen during our autotuning runs. Also, if we can fit a smooth curve (or surface) to our optimal parameters (which will be explored later) then this curve can be coded into Lua to provide parameters for matrices with various sizes. This provides a much more powerful and flexible interface than a simple configuration file allows.

Therefore, we have two separate problems to tackle. First, we need to use OpenTuner, or some other method such as a grid-sweep, to find optimal parameters for a range of matrix sizes. Second, we would like to fit a function to these points (so that a good value of the block size can be returned for any matrix size) before writing Lua code to implement this function. From here, the software can simple call the Lua script to obtain the parameters for any input matrix. This second issue is rather difficult to automate: there are a variety of different functions needed to fit the various behaviours we observe the block size showing.

For the first issue, performing a grid-sweep over all block sizes for a variety of matrix sizes is very simple and the optimal parameters can easily be stored in a csv file. In order to use OpenTuner there is a small amount of extra work required: we must define a Python class with the following 3 functions

- `manipulator` – to define the search space,
- `run` – to run our routine with the current parameters, and,
- `save_final_config` – to save the final parameters.

In `manipulator` we simply define the tile size as a multiple of 8 (the width of the vector units on most CPUs) between 80 and 520. Within `run` we simply run the routine we are currently investigating with the current block size and take the average time over 5 runs (where 5 is actually a user-chosen parameter). Finally in `save_final_config` we output the "optimal" block size in JSON format. A wrapper script collects the JSON files for various matrix sizes into a single CSV file.

The actual optimization is taken care of by OpenTuner itself, meaning that there is nothing else to do but to limit the number of tests performed using the command-line argument `test-limit=x`. We can also define multiple initial guesses for the optimal block size using the command-line argument `seed-configuration=f` where `f` is a JSON file containing, for example, `{"blocksize": 256}`. We have defined a few initial guesses for the powers of 2 etc.

4 Curve fitting

Once we have found the optimal block sizes for a variety of matrix sizes (1000 to 30000 in steps of 1000 within our experiments) we would like to fit a curve to the resulting data, which can then be coded in Lua. Using this fitted curve we can then return near-optimal parameters for any input matrix, regardless of its size.

In this section we will show the variety of behaviours that the optimal block size can take as the matrix size changes, propose a number of models to fit the resulting data, and describe how the parameters of such models can be estimated. We found that the following four models can cover all of the routines we investigated.

- Constant models - Block size does not depend on the input matrix size.
- Linear models - Block size increases linearly with the input matrix size.
- Logarithmic models - Block size increases logarithmically with the input matrix size.
- Piecewise models - A combination of the other three models is required, for example a step function is a combination of constant models.

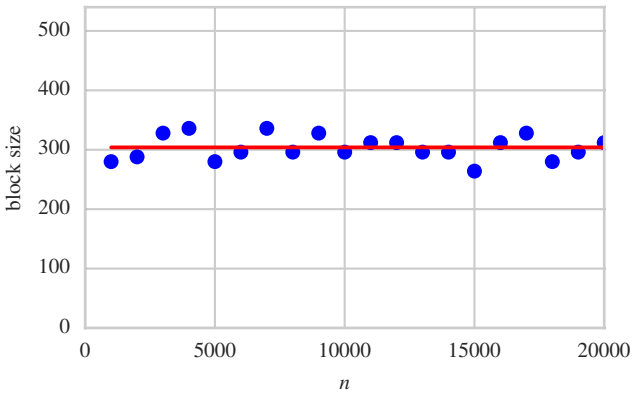

Figure 1: The optimal block size for DGEMM is independent of the matrix size so a constant model is most appropriate. On this architecture the constant is 304.

First we investigate the performance of DGEMM for matrix multiplication. As we can see from Figure 1, the optimal block size is largely independent of the matrix size. Therefore, we use the mean of all the values to return a constant block size: in this case the constant is 304. The reason for the lack of dependence on the matrix size is fairly straightforwards, since DGEMM is very arithmetic intensive, involving only fused multiply-adds, the block size is affected only by the memory hierarchy and cache size.

Next we investigate DGETRF for LU factorization in Figure 2. In this experiment it is clear that a linear model provides a much better fit for the data. Doing a least-squares fit gives an intercept of 92.25 and a gradient of 0.008.

In Figure 3 we observe that the block size for SPOTRF (Cholesky factorization in single precision) increases logarithmically with the matrix size. The least-squares fit of a

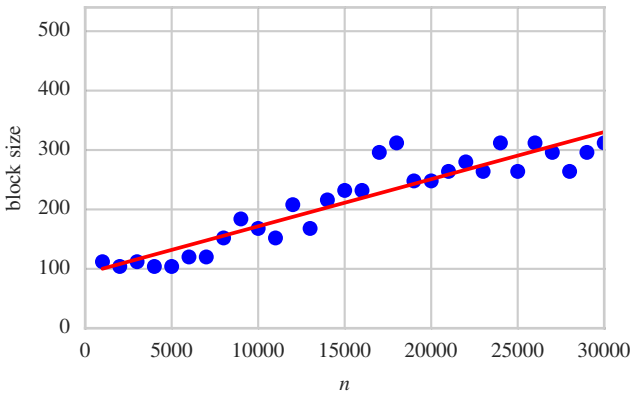

Figure 2: The optimal block size for DGETRF appears to depend linearly on the matrix size. In this model the intercept is 92.25 with a gradient of 0.008.

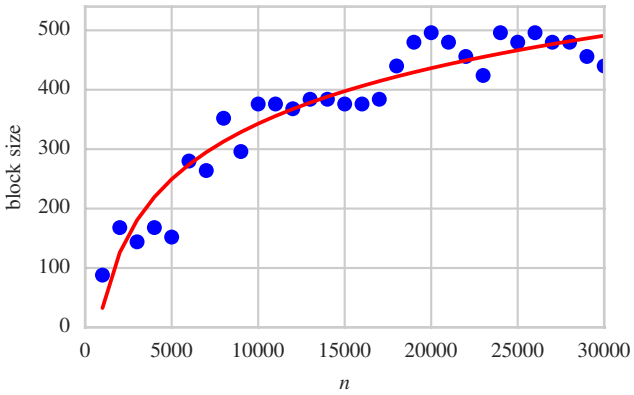

Figure 3: The optimal block size for SPOTRF appears to depend logarithmically on the matrix size. The least-squares fit provides the block size function $y = -898 + 135\log(x)$.

model $y = a + b\log(x)$ on a set of k data points (x_i, y_i) gives the parameters

$$b = \frac{k\sum y_i \log(x_i) - \sum y_i \sum \log(x_i)}{k\sum \log(x_i)^2 - (\sum \log(x_i))^2},$$
$$a = \frac{\sum y_i - b\sum \log(x_i)}{k}.$$

Fitting this model to our data gives the model $y = -898 + 135\log(x)$.

Finally, by looking at the performance of DPOTRF (Cholesky factorization in double precision) in Figure 4, we see an example of a routine where a piecewise function is the best choice to model the data. In this particular case a step function is an appropriate model, though there are other routines where initially the function is logarithmic before settling at some constant value. In this particular step function the value changes when the matrix sizes are 5000 and 19000, whilst the three constants are 124, 291, and 487.

In summary there are a variety of different behaviours that the block size can exhibit depending upon the routine in question. Note that even performing the same routine in different precisions can lead to drastically differing behaviours (compare Figures 3 and 4).

As such, we can only recommend that each routine is treated independently and no initial assumptions are made about their behaviour. Furthermore, these results are only

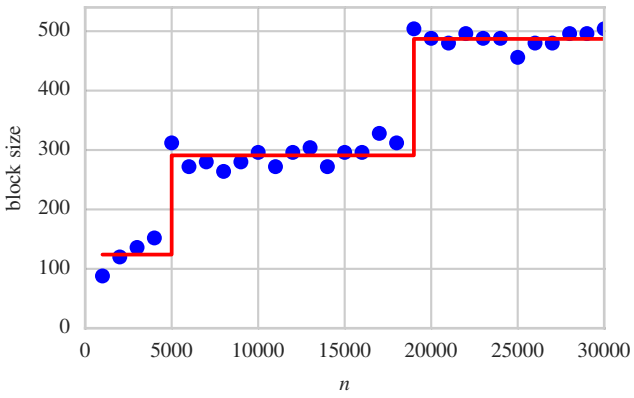

Figure 4: The optimal block size for DPOTRF appears to be a step function. The three constants in this step function are 124, 291, and 487.

measured on a Haswell NUMA node. Moving to more recent architectures, or the Intel Xeon Phi (codenamed Knights Landing) may lead to drastically different behaviours. Clearly, once these trends have been found, they need to be coded within Lua functions so that the software can find the appropriate block sizes at runtime.

5 Performance results

In this section we aim to demonstrate the improvements in performance that can be gained by applying the procedure described in the previous sections. Using a 2 socket NUMA node with Haswell cores (2 x Xeon(R) CPU E5-2650 v3, 2.30GHz) we ran the OpenTuner procedure to obtain the best block sizes, fit various curves to the resulting data as described in section 4, and then encoded these curves in Lua functions.

We will compare the performance of OpenMP PLASMA using the default parameters (i.e. taking a block size of 256) against our tuned version for four different routines: matrix multiplication, Cholesky factorization, LU factorization, and QR factorization. In each case we will show the performance using square matrices of various sizes in both single and double precision.

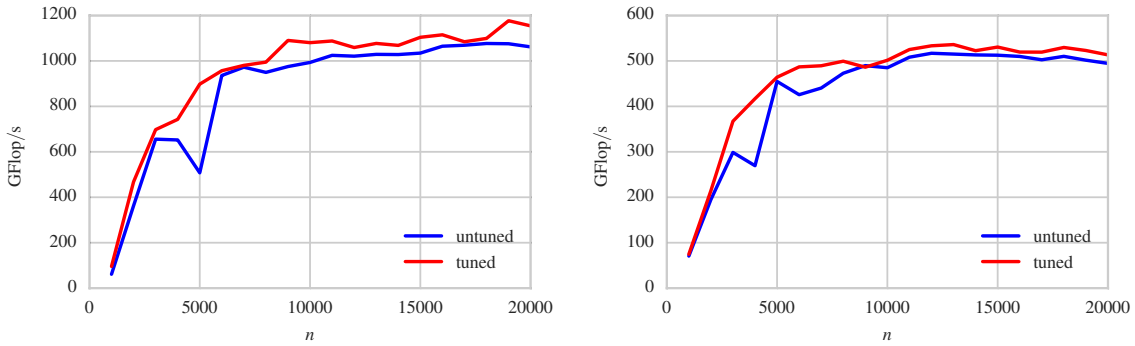

Figure 5: Tuned and untuned performance for the GEMM kernel. Single precision is on the left and double precision on the right.

First, in Figure 5, we look at the performance of GEMM (matrix multiplication) before

and after tuning. Single precision is on the left and double precision on the right. As we can see, the tuned version always performs at least as well as the untuned version and is often superior. Both versions converge to a similar performance, with the tuned version being marginally better.

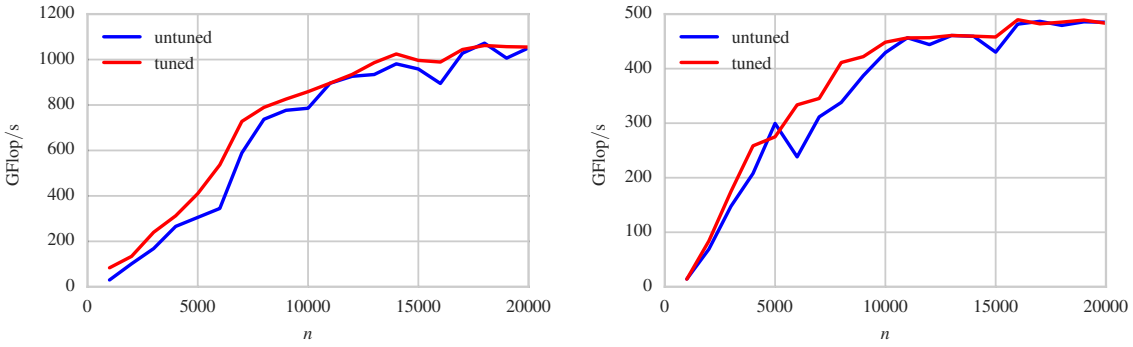

Figure 6: Tuned and untuned performance for Cholesky factorization. Single precision is on the left and double precision on the right.

Next we consider POTRF (the Cholesky decomposition) in Figure 6. As before we see that the tuned version is almost always superior to the untuned version and the two versions appear to converge to similar GFlop rates.

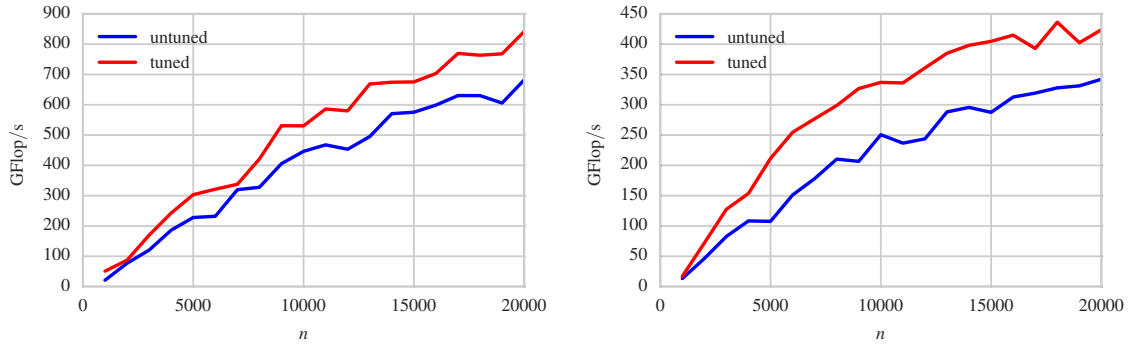

Figure 7: Tuned and untuned performance for LU factorization. Single precision is on the left and double precision on the right.

In Figure 7 we see the results of GETRF (the LU factorization). In this case we see that the tuned version always preferable and the gap between the two versions has widened. Indeed for the larger matrices in our tests we see a performance gap of over 100 GFlop/s between them, in both precisions.

Finally, Figure 8 shows the results for GEQRF (the QR factorization). In single precision arithmetic the tuned version is only slightly better than the untuned version. However, in double precision, the tuned version is far superior and is around 150 GFlop/s faster at one point. In both cases the two versions converge to a similar performance as the matrix size increases, though the tuned version is slightly faster.

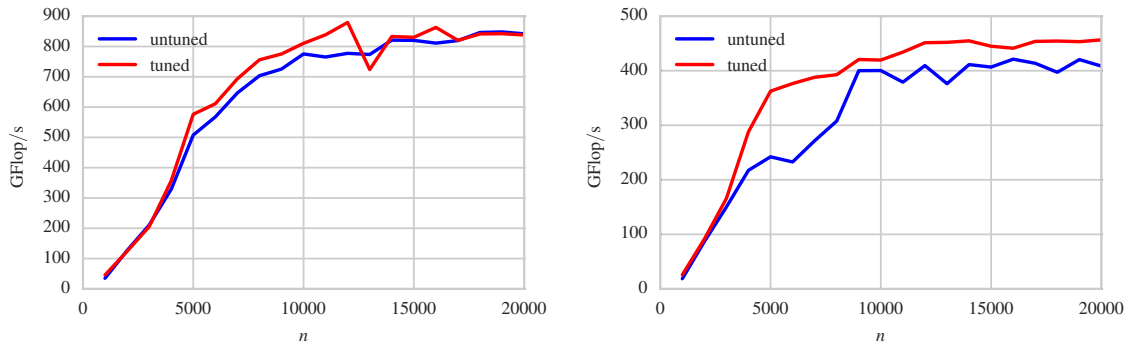

Figure 8: Tuned and untuned performance for QR factorization. Single precision is on the left and double precision on the right.

6 Conclusions

To conclude, we have described a process to move from a piece of software with no tuning whatsoever towards software which is highly tuned for a given architecture. Our process makes use of the OpenTuner optimization software to find optimal parameters, combined with data analysis and curve fitting techniques which can then be translated to the Lua scripting language.

As seen from our experiments with the PLASMA library for dense linear algebra, we can obtain results that are superior to untuned versions in almost every test case.

Future work in this area could focus on functions with multiple tuning paramers. In such functions it may be more difficult to fit a surface to the data found by OpenTuner. Some possible ways to tackle this may be to fit each parameter separately and use a copula (an idea borrowed from statistics), or to apply Gaussian process regression in this multi-dimensional setting.

References

[1] Emmanuel Agullo, Jim Demmel, Jack Dongarra, Bilel Hadri, Jakub Kurzak, Julien Langou, Hatem Ltaief, Piotr Luszczek, and Stanimire Tomov. Numerical linear algebra on emerging architectures: The PLASMA and MAGMA projects. *Journal of Physics: Conference Series*, 180(1), 2009.

[2] Jason Ansel, Shoaib Kamil, Kalyan Veeramachaneni, Jonathan Ragan-Kelley, Jeffrey Bosboom, Una-May O'Reilly, and Saman Amarasinghe. Opentuner: An extensible framework for program autotuning. In *International Conference on Parallel Architectures and Compilation Techniques*, Edmonton, Canada, August 2014.

[3] R. Miceli et. al. AutoTune: a plugin-driven approach to the automatic tuning of parallel applications. *Applied Parallel and Scientific Computing. PARA 2012. Lecture Notes in Computer Science*, 7782:328–342, 2013.

[4] Y. Oleynik, R. Mijakobić, I. A. Comprés Ureña, M. Firbach, and M. Gerndt. *Tools for High Performance Computing*, chapter Recent Advances in Periscope for Performance Analysis and Tuning, pages 39–51. Springer, 2014.